A new Notion of Useful Cache Block to Improve the Bounds of Cache-Related Preemption Delay*

Sebastian Altmeyer, Claire Burguière
Compiler Design Lab
Saarland University
66041 Saarbrücken, Germany
{altmeyer,burguiere}@cs.uni-saarland.de

Abstract

In preemptive real-time systems, scheduling analyses are based on the worst-case response time of tasks. This response time includes worst-case execution time (WCET) and context switch costs. In case of preemption, cache memories may suffer interferences between memory accesses of the preempted and of the preempting task. These interferences lead to some additional reloads that are referred to as cache-related preemption delay (CRPD). This CRPD constitutes a large part of the context switch costs. In this article, we focus on the computation of upper bounds on the CRPD using the concept of useful cache blocks (UCB). These are memory blocks that may be in cache before a program point and may be reused after it. When a preemption occurs at that point the number of additional cache-misses is bounded by the number of useful cache blocks. We tighten the CRPD bound by using a modified notion of UCB: Only cache blocks that are definitely cached are considered useful by our approach. As we show in this paper, the computed CRPD based on our notion, when used in combination with the bound on the WCET, delivers a safe bound on the execution time in case of preemption. Furthermore the modified definition simplifies the UCB computation for set-associative LRU and data caches. Experimental results show that our approach provides up to 90% tighter CRPD bounds.

1 Introduction

Hard real-time systems impose strict timing constraints. To prove that these constraints are met, timing analyses aim to derive safe upper bounds on a task's execution time. Being already a difficult problem in case of non-preemptive execution, it is even more challenging for preemptively scheduled tasks. Most approaches, to solve this, use separate computations of a task's execution time and the context switch costs. In [8], Lee et al. present a method to bound the context switch costs using the concept of useful cache blocks (UCB). This concept has been extended by several research groups (e.g. [13, 17]). A useful cache block at program point P is a memory block that *may* be in cache at P and *may* be subsequently reused. When such a memory block is evicted due to preemption, additional cache-misses may occur. The additional reloads due to these misses contribute to the context switch costs (CSC) and are referred to as cache-related preemption delay (CRPD). Finally, to estimate a task's response time, the corresponding schedulability analysis adds these costs to the bound of the execution time as often as preemption may occur.

An over-approximation of the cache-content (memory block *may* be in cache) is used by former UCB analyses to derive a safe upper bound on the context switch costs. Timing analysis, however, also uses an over-approximation of the cache-content to predict the number of cache-misses (and an under-approximation to predict the number of cache-hits). During schedulability analysis, thus, some cache-misses may be counted twice. So, treating timing analysis and CSC analysis separately, the over-approximation on both sides accumulates and introduces a high degree of pessimism. Figure 1 illustrates an example showing the over-approximation of the execution time (a), the context switch costs (b) and the combined over-approximation (c).

In our approach, we stick to the concept of useful cache blocks, but provide a more precise estimation: A memory block is only considered useful, if it *must* be in cache. We refer to such a UCB as a *definitely-cached UCB* (DC-UCB) As seen before, timing analysis and prior UCB definition count some potential cache reloads twice. So, we compute an approximation of the context switch costs[1] regard-

*This work was supported by ICT project PREDATOR in the European Community's Seventh Framework Programme under grant agreement no. 216008, by Transregional Collaborative Research Center AVACS of the German Research Council (DFG) and by ARTIST DESIGN NoE.

[1]The CSC are given by the CRPD plus a constant overhead. This overhead is due to pipeline flush/reload costs and depends on the

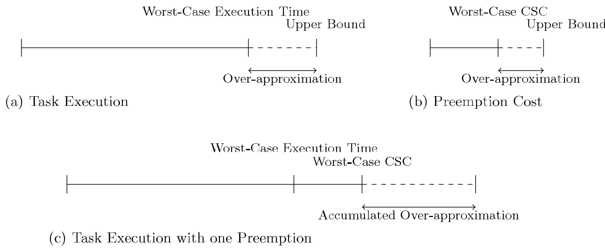


Figure 1: Over-approximation of WCET and context switch cost.

ing the corresponding timing analysis and using an under-approximation of the cache as shown in Figure 2. The computed context switch costs may be in fact an under-approximation of the real costs (b). However, they are safe if combined with the derived execution-time bound (a): The over-approximation of the execution time subsumes the under-approximation of the context switch cost computation (c).

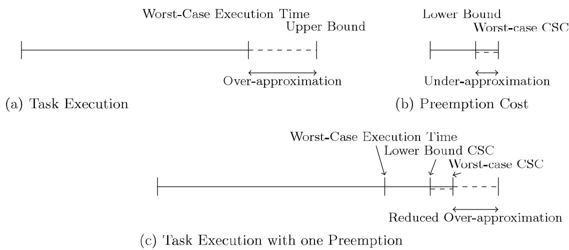


Figure 2: Over-approximation of WCET, under-approximation of the context switch cost, safe over-approximation of overall task execution time under preemption

Thus, our approach strongly improves the bound on the cache-related preemption delay. Furthermore, a DC-UCB analysis for the LRU replacement policy as well as for data caches (which caused complications to prior work) comes for free; cache analysis for these caches already exist and can be used by our analysis.

The paper is structured as follows. The next section provides basic concepts of caches and timing analysis. In Section 3, we introduce an adapted notion of useful cache block (definitely-cached UCB), prove the soundness of our approach and present the corresponding program analysis. Section 4 presents related work and the evaluation of our approach is given in Section 5. Finally, Section 6 concludes the paper.

target architecture. Its computation is out of scope of this paper.

2 Timing Analysis

This section introduces basic concepts and notations of caches and timing analysis. Both are needed as background information for main Section 3.

2.1 Caches

Caches are fast and small memories storing frequently used memory blocks to close the increasing performance gap between processor and main memory. They can be implemented as data, instruction or unified caches. Caches are divided into *cache-lines*. A *cache-line* is the basic unit to store *memory blocks*, i.e., blocks of *line-size* l contiguous bytes of memory. The set of all memory blocks is denoted by M. The cache-size s is, thus, given by the number of cache-lines c times line-size l. A set of n cache-lines forms one *cache-set*, where n is the *associativity* of the cache and determines the number of cache-lines a specific memory block may reside in. The number of sets is given by c/n and the cache-set memory block b maps to by: $b \ mod(c/n)$. Special cases are direct-mapped caches ($n = 1$) and fully-associative caches ($n = c$). In the first case, each cache-line forms exactly one cache-set and there is exactly one position for each memory block. In the second case, all cache-lines together form one cache-set and all memory blocks compete for all positions. If the associativity is higher than one, a *replacement policy* has to decide in which cache-line of the cache-set a memory block is stored, and, in case all cache-lines are occupied, which memory block to evict. In this paper, we concentrate on direct-mapped caches and on the LRU replacement policy.

2.2 Timing Analysis

Several static timing analyses exist to bound the worst-case execution time of tasks (see [4] for an overview). These tasks under examination are represented as control flow graphs (CFG). Nodes of a CFG are basic blocks; maximal sequences of instructions with exactly one entry and one exit point. If one instruction of a basic block is executed, so are all others. The edges of a control flow graph represent the possible control flow.

Within a typical timing analysis, a value-analysis first derives effective addresses of memory accesses as well as values for registers and memory cells. So, it supports the loop-analysis deriving bounds on the maximal number of loop-iterations. In the next step, an upper bound on the execution time of each basic block is derived using a low-level analysis simulating the target processor's behavior. A part of this low-level analysis is the cache-analysis, which aims to classify memory accesses – for which the value analysis derived the effective addresses – into cache-hits and cache-misses. Note that we describe a cache-analysis as used in

our approach in the next subsection.

In the last step, the above information is combined to find the path within the control flow graph with the highest execution time bound.

An important issue, which we have to mention here, is the treatment of unclassified memory accesses. Depending on the target architecture, it may be reasonable to assume that a cache-miss always leads to a higher execution time than a cache-hit. Therefore, all memory accesses that could not be classified as cache-hit are treated as cache-miss. As Lundqvist et al. [10] have shown, this, however, is not true in general for modern architectures. So-called *timing anomalies*, mainly induced by processor features with long term effects, such as buffers or prefetch queues, may cause a locally faster execution, i.e., a cache-hit (instead of a cache-miss) to lead to a globally higher execution time. A separate computation of the timing bound and the context switch costs, thus, always implies a target processor without timing anomalies.

Throughout this paper, we will use the following notation: A control flow graph is given by $CFG = (V, E, s, e)$ where $V = \{B_1, \ldots B_n\}$ denotes the set of basic blocks B_i of a program and $E \subseteq V \times V$ the corresponding set of edges connecting them. The start node is denoted by s and the end node by e. Furthermore, we denote instruction j of basic block i with B_i^j. We use the partial function $Access_D(B_i^j)$ to denote the memory block of a possible data access during execution of instruction B_i^j. In case no data memory is accessed, $Access_D(B_i^j)$ is not defined. Furthermore, we use $Access_I(B_i^j)$ as the address of the instruction. Note that we omit the index D, I in case no distinctions need to be made or when it is determined by the context.

2.3 Cache Analysis

As a part of a timing analysis, a cache analysis aims to statically predict the cache behavior during the execution of a task. For this reason, the analysis classifies memory accesses as cache-hits or cache-misses. Due to input-dependent cache behavior and an unknown initial cache state, this classification is not complete, i.e., not all memory accesses can be classified. To circumvent this problem, the concept of *may* and *must* information has been introduced to bound the cache contents from above and below. The may-cache contains all memory blocks that may be cached at a given program point, i.e., where the analysis is unable to prove that the memory block is not cached. Vice versa, the must-cache at a program point contains all memory blocks the analysis can prove to be cached.

Several different cache-analyses have been proposed [3, 6, 9, 12, 23, 18], either for direct-mapped or n-way associative LRU caches, and for instruction- or data-caches. Our approach is generic in the chosen cache analysis. All we need is the result of a preceding cache analysis, i.e., a safe classification of memory accesses. In the following, we assume that for each instruction B_i^j, a set

$$Must_Cache(B_i^j) \subseteq M$$

is given that contains all memory blocks that are cached before execution of B_i^j. The cache-analysis delivering this classification must be the same as used by the overall timing analysis in order to ensure soundness of the approach.

In its simplest form, a must-cache analysis for direct-mapped caches is implemented as a program analysis keeping an abstract cache state at each program point. Such an abstract cache state contains for each cache-set s either a memory block or the symbol $\top$ indicating that the content of s could not be predicted. Encountering an access to a memory block m, the analysis stores m at the corresponding position in the abstract state (while replacing the prior content; see Figure 3 (a)). Two abstract cache states are combined by keeping equal contents within both states and replacing conflicting contents by $\top$ (Figure 3 (b)). In our approach, the must-cache must be given as a set. This set can be simply derived by collecting all elements contained in the abstract cache-sets. For data caches, the value analysis statically derives effective addresses of memory accesses (the memory block of an instruction is given by its address). If the value analysis, however, fails for some accesses, the precision of the cache-analysis suffers. Even if the analysis is able to give a range for an access, for instance an access to an array component, the must-cache analysis can often not predict which block is cached.

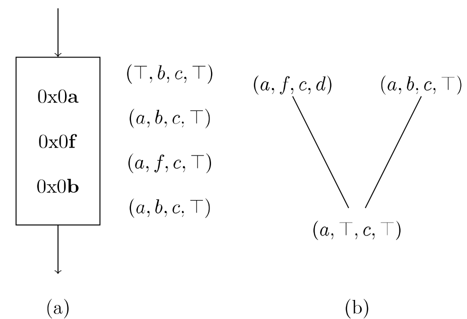


Figure 3: Must-cache analysis: (a) straight-line code and (b) control flow merge

3 Cache-Related Preemption Delay

Static computation of upper bounds on cache-related preemption delay (CRPD) in most cases relies on the analysis

of useful cache blocks (UCB). The basic concept behind useful cache blocks is the determination of cache-blocks that may cause additional cache-reloads due to preemption. Lee et al. [8] identified two properties a memory block must have to be considered a useful cache block at program point P; a) it *may* be in cache at P and b) it *may* be reused on at least one control flow path starting at P. If the task is preempted at this point, the number of additional cache-misses due to preemption, thus, is bounded by the number of useful cache blocks. By selecting the program point with the maximal number of UCBs, an upper bound on the overall CRPD for one preemption of the task is given by this maximal number. Several research groups extended this concept, all relying on the same definition of useful cache blocks (see Section 4).

In contrast to these approaches, we introduce the notion of *definitely-cached UCB* as follows:

Definition (Definitely-cached UCB)
A definitely-cached UCB (DC-UCB) is a memory block which:

a) **may** be reused on at least one control flow path starting at P,

b) **must** be cached at P and along the path to its reuse.

Prior analyses on cache-related preemption delay define a cache block to be useful at program point P if it b) *may* be cached at P. We replace *may* by *must* and reduce the set substantially. Furthermore, we require it to be cached along the path to its reuse. If it is not in cache all along this path, it may have been removed on it and, thus, does not lead to an additional cache-miss due to preemption. We can use a cache-analysis – as presented before – to check for condition b). So, we can also establish a DC-UCB analysis for data caches.

In the following, we show that this definition leads to a safe but reduced over-approximation of the additional CRPD. Afterwards, we present the analysis of DC-UCBs following our definition and discuss further extensions.

3.1 Soundness

The combination of over-approximation in the timing analysis and in the conventional notion of useful cache blocks counts some potential cache reloads twice. The adapted definition excludes these reloads: So, they are only counted as part of the execution time bound. The context switch costs, thus, may be an under-approximation of the real costs but are a sound over-approximation when combined with the results of the timing analysis. This can be seen by comparing (i) the overall number of cache-misses that might occur during task execution with (ii) the number of cache-misses the analyses (timing analysis and DC-UCB analysis) take into account. If our approach accounts for at least the number of actual cache-misses, i.e., the number of cache-misses that occur during the execution of the task, it can be considered to derive a safe over-approximation.[2]

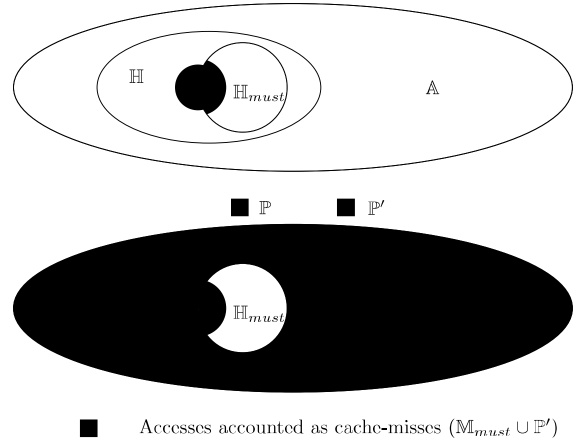


Figure 4: Set of memory accesses $\mathbb{A}$, set of cache-hits $\mathbb{H}$, set of cache-hits approximated by must-cache $\mathbb{H}_{must}$, set of cache-misses due to preemption $\mathbb{P}$ and set of cache-misses $\mathbb{P}'$ derived by our analysis. The dotted part in the second graph denotes all accesses taken into account as cache-misses by our approach.

Consider the set $\mathbb{A}$ of memory accesses that occur during the non-preemptive execution of a task. These accesses are either cache-hits $\mathbb{H}$ or cache-misses $\mathbb{M}$. By construction, the must-cache analysis classifies a subset of these accesses as cache-hits $\mathbb{H}_{must} \subseteq \mathbb{H}$. All other accesses are taken into account as cache-misses $\mathbb{M}_{must} = \mathbb{A} \setminus \mathbb{H}_{must}$:

$$\mathbb{H} \cup \mathbb{M} = \mathbb{A} = \mathbb{H}_{must} \cup \mathbb{M}_{must}$$

The set of additional cache-misses due to preemption is denoted by $\mathbb{P} \subseteq \mathbb{H}$. By definition the set $\mathbb{P}'$ obtained by our analysis is an over-approximation of $\mathbb{P}$ restricted to elements of the set $\mathbb{H}_{must}$:

$$\mathbb{P}' \supseteq \mathbb{P} \cap \mathbb{H}_{must}$$

By removing this set $\mathbb{P}'$ from the set of cache-hits classified by the must cache, we get:

$$\mathbb{H}_{must} \setminus \mathbb{P}' \subseteq \mathbb{H}_{must} \setminus (\mathbb{P} \cap \mathbb{H}_{must}) = \mathbb{H}_{must} \setminus \mathbb{P}$$

Since $\mathbb{H}_{must} \subseteq \mathbb{H}$, the set of cache-hits classified by our method is a subset of the actual set of cache-hits, we get:

$$\mathbb{H}_{must} \setminus \mathbb{P}' \subseteq \mathbb{H} \setminus \mathbb{P}$$

[2]Remember that using UCBs to compute context-switch costs is restricted to processors not exhibiting timing anomalies.

This shows that our approach under-approximates the set of cache-hits under preemption: The number (i) of actual cache-misses, given by $(\mathbb{M} \cup \mathbb{P})$, is a subset of the number (ii) of cache-misses taken into account by our analysis, given by $(\mathbb{M}_{must} \cup \mathbb{P}')$. By this, we have shown that our approach safely bounds the cache-related preemption delay when combined with an upper bound on the WCET. Figure 4 illustrates the different sets and their relation to each other.

3.2 Program Analysis

To determine the set of definitely-cached UCBs, we use a backward program analysis on the control flow graph. A memory block m is added to the set of DC-UCBs of instruction B_i^j, if m is element of the must-cache at B_i^j and if instruction B_i^j accesses m. The domain of our analysis is the powerset domain on the set of memory blocks M:

$$\mathbb{D} = 2^M$$

The following two equations determine the flow information before (DC-UCB$_{in}$) and after (DC-UCB$_{out}$) instruction B_i^j:

$$\text{DC-UCB}_{in}(B_i^j) = gen(B_i^j) \cup (\text{DC-UCB}_{out}(B_i^j) \setminus kill(B_i^j)) \quad (1)$$

$$\text{DC-UCB}_{out}(B_i^j) = \bigcup_{successor B_k^l} \text{DC-UCB}_{in}(B_k^l) \quad (2)$$

where the gen/kill sets are defined as follows:

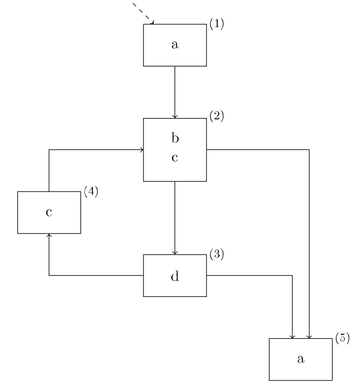
$$gen(B_i^j) = \begin{cases} \{Access(B_i^j)\} & \text{if } Access(B_i^j) \in Must_Cache(B_i^j) \\ \emptyset & \text{otherwise} \end{cases} \quad (3)$$

$$kill(B_i^j) = M \setminus Must_Cache(B_i^j) \quad (4)$$

The direction of the analysis is backward. Equation (2) combines the flow information of all successors of instruction B_i^j. Equation (1) represents the update of the flow information due to the execution of the instruction. First, all memory blocks not contained in the must-cache at B_i^j are removed from the set of DC-UCBs (4) – only a memory block that is element of the must-cache all along the way to its reuse is considered useful by our definition. Then, the accessed memory block of instruction B_i^j is added in case it is contained in the must-cache at the instruction (3).

Using these equations, the set of UCBs can be computed via fix-point iteration (see [2]). The initial values at instruction B_i^j are defined by DC-UCB$_{in}(B_i^j)$ = $gen(B_i^j)$ and DC-UCB$_{out}(B_i^j) = \emptyset$. Note that Equation (1) fits the kill/gen scheme to ensure that the fix-point iteration will derive the smallest solution to the set of equations. In terms of program-analyses, *minimal fix-point* solution (MFP) equals the *merge-over all paths* solution (MOP). However, we can also simplify Equation (1) as follows:

$$\text{DC-UCB}_{in}(B_i^j) = gen(B_i^j) \cup (\text{DC-UCB}_{out}(B_i^j) \cap Must_Cache(B_i^j)) \quad (5)$$



I	Must-Cache	DC-UCBs
B_1^1	$(_,_,_,_)$	$(_,_,_,_)$
B_2^1	$(a,_,_,_)$	$(a,_,_,_)$
B_2^2	$(a,b,_,_)$	$(a,_,_,_)$
B_3^1	$(a,b,c,_)$	$(a,_,c,_)$
B_4^1	(a,b,c,d)	$(a,_,c,_)$
B_5^1	$(a,_,c,_)$	$(a,_,_,_)$

Figure 5: Example Control flow graph and corresponding content of the must-cache.

Consider the control flow graph given in Figure 5. The letters within basic blocks denote the memory blocks accessed by the instructions. The table shows the content of the must-cache assuming a direct-mapped data cache of size 4 and the obtained DC-UCBs. Table 1 lists the equations our analysis uses and Table 2 the steps and the resulting sets of the fix-point iteration. Note that we use the simplified Equation (5). Memory block a is cached before and reused at instruction B_5^1. So a is a DC-UCB before B_5^1 and contained in the initial state of DC-UCB$_{in}(B_5^1)$ (Table 2, $i = 0$). The same holds for memory block c and instruction B_4^1. In the next step of the computation ($i = 1$), a is furthermore considered a DC-UCB at B_5^1's predecessors B_2^2 and B_3^1, and c at B_4^1's predecessor B_3^1. The equations provided in Table 1 are iteratively applied until a fix-point on the set of DC-UCBs is reached.

Note that prior UCB analyses employ two program analyses, one to check for condition a) memory block may be cached at position P, the other for condition b) memory block may be reused. If we adhere to this structure, we can see the cache analysis as presented in the Section 2.3 as one program analysis (a) – the analysis which we presented above as the other one (b).

3.3 LRU Caches

Although the DC-UCB analysis was introduced using an example for a direct-mapped cache, the analysis is valid in the same manner for set-associative or fully-associative LRU

B_i^j	DC-UCB$_{in}(B_i^j)$
B_1^1	DC-UCB$_{out}(B_1^1) \cap \emptyset$
B_2^1	DC-UCB$_{out}(B_2^1) \cap \{a\}$
B_2^2	DC-UCB$_{out}(B_2^2) \cap \{a, b\}$
B_3^1	DC-UCB$_{out}(B_3^1) \cap \{a, b, c\}$
B_4^1	$\{c\} \cup$ (DC-UCB$_{out}(B_4^1) \cap \{a, b, c, d\}$)
B_5^1	$\{a\} \cup$ (DC-UCB$_{out}(B_5^1) \cap \{a, c\}$)
B_i^j	DC-UCB$_{out}(B_i^j)$
B_1^1	DC-UCB$_{in}(B_2^1)$
B_2^1	DC-UCB$_{in}(B_2^2)$
B_2^2	DC-UCB$_{in}(B_3^1) \cup$ DC-UCB$_{in}(B_5^1)$
B_3^1	DC-UCB$_{in}(B_4^1) \cup$ DC-UCB$_{in}(B_5^1)$
B_4^1	DC-UCB$_{in}(B_2^1)$
B_5^1	$\emptyset$

Table 1: Equations of DC-UCB analysis

	DC-UCB$_{in}$					
i	B_1^1	B_2^1	B_2^2	B_3^1	B_4^1	B_5^1
0	$\{\}$	$\{\}$	$\{\}$	$\{\}$	$\{c\}$	$\{a\}$
1	$\{\}$	$\{\}$	$\{a\}$	$\{a, c\}$	$\{c\}$	$\{a\}$
2	$\{\}$	$\{a\}$	$\{a\}$	$\{a, c\}$	$\{c\}$	$\{a\}$
3	$\{\}$	$\{a\}$	$\{a\}$	$\{a, c\}$	$\{a, c\}$	$\{a\}$

	DC-UCB$_{out}$					
	B_1^1	B_2^1	B_2^2	B_3^1	B_4^1	B_5^1
0	$\{\}$	$\{\}$	$\{\}$	$\{\}$	$\{\}$	$\{\}$
1	$\{\}$	$\{\}$	$\{a\}$	$\{a, c\}$	$\{\}$	$\{\}$
2	$\{\}$	$\{a\}$	$\{a, c\}$	$\{a, c\}$	$\{c\}$	$\{\}$
3	$\{a\}$	$\{a\}$	$\{a, c\}$	$\{a, c\}$	$\{a, c\}$	$\{\}$

Table 2: Fix-point iteration to derive the set of DC-UCBs

caches. The DC-UCB analysis only relies on the information about the currently accessed memory blocks and on an under-approximation of the cache-content, i.e., the must-cache. The structure of the cache, however, is completely hidden in the set $Must_Cache(B_i^j)$, which is obtained by the cache analysis as part of the WCET analysis. Therefore, if the WCET analysis is able to handle LRU caches, so is the DC-UCB analysis.

3.4 Data Caches / Handling Array Accesses

If the addresses of the memory accesses are known statically, the same DC-UCB analysis as for instruction caches can be applied. For some data accesses, however, the value analysis is unable to derive precise addresses statically. Consider an array access within a loop, for instance. The actual address of the access changes each iteration. The value analysis, thus, derives a range bounding the address of the access instead of a single value. In addition to $Access(B_i^j)$ which denotes the address of the access, $Range(B_i^j)$ denotes the length of it. In case no range is needed, $Range(B_i^j) = 1$.

To handle imprecise information about the addresses of memory accesses – and to enable a general DC-UCB analysis for data caches[3] – we have to adapt the analysis as follows. The domain is given by a multiset:

$$\mathbb{D} = 2^{(M,I)}$$

where I denotes the length of the range given in the number of memory blocks. An access whose effective address is bounded by a memory range is only considered a cache-hit if the whole range is definitely cached. Therefore, the access is also only considered a definitely-cached UCB in this case (6). Elements are removed from the set of DC-UCBs only if no memory block is definitely cached (7). Otherwise, if one memory block is cached and this memory block is removed due to preemption, it may cause an additional cache-miss. The corresponding kill/gen sets are specified as follows:

$$gen(B_i^j) = \begin{cases} \{(Access(B_i^j), Range(B_i^j))\} \\ \quad \text{if } \forall_{l=0}^{l<Range(B_i^j)} (Access(B_i^j) + l) \in Must_Cache(B_i^j) \\ \emptyset \text{ otherwise} \end{cases} \tag{6}$$

$$kill(B_i^j) = \{(Access(B_k^m), Range(B_k^m)) | \\ \forall_{l=0}^{l<Range(B_k^m)} (Access(B_k^m) + l) \notin Must_Cache(B_i^j)\} \tag{7}$$

An example of how the extension of the analysis works is shown in Figure 6. Assume that a, b and c are sequentially ordered memory blocks. Because the whole range of memory blocks, the last instruction may access, are cached, the timing analysis considers the data access of this instruction as a cache-hit. For the same reason, the definitely-cached UCB analysis adds $(a, 3)$ to the set of DC-UCBs. Only before the execution of the first and after execution of the last instruction, $(a, 3)$ is not a definitely-cached UCB; at every other position, a cache block evicted due to preemption may cause an additional cache-miss at the last memory access.

3.5 From UCB to CRPD

The final step of the DC-UCB analysis is the computation of the cost the schedulability analysis has to add to the bound of the WCET. For this, we first identify the program point with the highest number of DC-UCBs. Then, we multiply this number by the time needed to reload a memory block, to obtain an upper bound on the additional cost of the whole task. Note that this cost is not an upper bound on the actual CRPD, but on the CRPD that we have to take into account when used in combination with an upper bound on the WCET.

[3]If the value analysis can not even derive a memory range for an access, timing analysis treats this memory access as a cache-miss. It, therefore, does not contribute to the bound on the CRPD.

Memory accesses	Must-Cache	DC-UCBs
↓	$\emptyset$	$\emptyset$
0x0**a**		
↓	$\{a\}$	$\{(a,3)\}$
0x0**b**		
↓	$\{a,b\}$	$\{(a,3)\}$
0x0**c**		
↓	$\{a,b,c\}$	$\{(a,3)\}$
0x0**a**-0x0**c**		
↓	$\{a,b,c\}$	$\emptyset$

Figure 6: Data Cache DC-UCB analysis handling memory ranges; a sequence of memory accesses, cache content and sets of DC-UCBs.

4 Related Work

The problem caused by cache-interferences within a preemptive system can be solved using diverse approaches. It can be circumvented by partitioning cache memory, it can be taken into account as part of the worst-case execution time and it can be analyzed separately by estimating the cache-related preemption delay.

Some works [11, 24] use a simple method to avoid interferences of tasks on the cache by partitioning it and assigning each task a small part. Although there is no interference on the cache, the performance of each task may suffer by its decreased cache size and substantial code-changes are needed.

Schneider [15] includes the impact of cache-interferences in the WCET analysis. He assumes preemption at each program point to compute a safe upper bound on the execution time under preemption. However, just because preemption is assumed everywhere, the analysis largely overestimates a task's worst-case execution time. The method we use analyzes cache-interferences separately from the worst-case execution time analysis by estimating the cache-related preemption delay.

The cache-related preemption delay denotes the preemption cost due to the reload of cache blocks that are evicted during execution of a preempting task. This number is then included in the response time computation during scheduling analysis [5]. The CRPD is computed by multiplying the time needed to reload a memory block with the number of additional cache-misses due to preemption [1]. Two ways can be used to bound this number: computing the number of useful cache blocks [7] or computing the number of evicting cache blocks (ECB) [22]. An ECB is a memory block which is used by the preempting task and which may evict a memory block from the cache. An upper bound on the CRPD can be derived using directly the number of UCBs (or ECBs): All UCBs are evicted by the execution of the preempting task (or each ECB evicts a memory block that is reused later by the preempted task). A refinement of the upper bound on the CRPD is given by the intersection of UCBs and ECBs, as shown in [13, 19].

In this article, we focus on the analysis of useful cache blocks. So, in the rest of this section, different methods to obtain the sets of UCBs are presented.

The basic concept of useful cache block has been introduced by Lee et al. [8]. They use two data flow analyses to derive them: First to compute for each node in the CFG the set of cache blocks that may be cached (reaching cache blocks), and second, to compute the set of cache blocks that are potentially reused after this node (live cache blocks). The intersection of both sets delivers the set of UCBs. They represent cache contents by sets of memory blocks for all cache sets. Negi et al. [13] enhanced this computation by integrating a tighter cache representation that does not merge cache contents. However, as the sets representing the cache are larger and grow faster, the complexity of the UCB computation is significantly higher than for the previous representation. Staschulat et al. [18] combined these two representations to obtain less complexity than the second one but more precision than the first. Their results represent a scalable tradeoff between always merging cache content and never merging it. In addition, Staschulat et al. proposed to refine the CRPD computation in case of multiple preemption by taking into account the preempting task execution [16]. In [20], Tan et al. studied the case in which preemptions are nested (preempting task is preempted). The difference to our work is given by their notion of useful cache blocks. It includes cache reloads that are already accounted for by the timing analysis. Since we only concentrate on the notion of UCB, these extensions can be seen orthogonal to our approach, i.e., they could be also applied to our analysis.

In case multiple preemptions of the same task are possible, the highest CRPD bounds are considered. To compute a bound for the i. preemption, an order on the number of UCBs is used that considers each set of UCBs as often as the corresponding program point may be executed [7]. For example, the highest CRPD is typically obtained in a loop body. In that case, the bound corresponding to this point is considered as many times as it may be executed. This extensions can be used in same manner using our notion of DC-UCBs.

Furthermore, former work mainly present the UCBs computation for direct-mapped instruction caches. Two of them [8, 18] propose extensions to include set associative cache with LRU replacement. Data caches restricted to static addressing have also been taken into account by Lee et al.[7]. As far as we know, no work focuses on UCB computation for dynamic addressing of data caches. This dynamic addressing has been handled by Ramprasad et al. in [14]. However, they do not compute UCBs; they use cache access pattern (cache lines access chains) to derive the n most expensive preemption points. While our approach handles

instruction and/or data-caches, their approach is restricted to data caches.

In this paper, we use results of the cache analysis to refine the computation of the number of misses due to preemption. We propose a safe modification of the UCB definition (definitely-cached UCB). Our approach can be combined to the ECB computation to refine the resulting CRPD (as presented by [13, 19]).

5 Evaluation

In this section, we evaluate the precision and runtime of our approach. Since we concentrate on the improvement solely caused by our new notion of useful cache blocks, we compare it with Lee's approach [8] based on the original notion. Note that Negi et al. as well as Staschulat and Ernst proposed extensions to improve the precision of Lee's UCB analysis by using a more precise representation of the cache-states. Since our paper focuses on the basic notion of UCB and not on the cache-state representation, we compare our approach with the original UCB approach. A comparison of our work with Negi et al. as well as Staschulat and Ernst extensions is future work.

As target architecture, we selected an Arm7 processor[4] with direct-mapped instruction cache of size a) 1kB, line size 8 Byte, b) 4kB, line size 16 Byte and c) 8kB, line size 8 Byte. The Arm7 features an instruction size of 4 Byte.

We employed a selection of the Mälardalen WCET benchmark suite[5]. Table 3 shows this selection, the number of instructions of each task and ratios of task-size to cache-size. We compiled these tests using a gcc cross-compiler.

To tighten the bound on the worst-case execution time, a technique called virtual inlining and loop unrolling [21] is usually applied. This technique artificially increases the control flow graph to distinguish between different loop iterations and function calls. It especially improves the must-cache analysis, i.e., more memory accesses will be classified as cache-hits. For the following test-cases, we also employed virtual unrolling and virtual inlining in order to derive realistic results. Note that by improving the precision of the must-cache, we increase the sets of DC-UCBs and so, we can only decrease the improvement we obtain.

Tables 4, 5 and 6 show the results for the different cache-sizes. The average number of (DC-)UCBs per instruction (Column 2 and 3) can be seen as an indicator of the CRPD in case preemption is restricted to a given set of preemption points. Column 4 shows the average improvement, i.e., the sum over the improvements of each instruction divided by the number of instructions. Column 5 and 6 show the maximal number of (DC-)UCBs for the given task and the last column the improvement. The maximal number of (DC-)UCBs multiplied by the cache reload cost constitutes an overall upper bound on the CRPD of the whole task (when combined with an upper bound of the WCET).

	average			upper bound		
	#UCB	#DC-UCB	diff	#UCB	#DC-UCB	diff
bs	13.6	1.4	52%	24	5	79%
bsort100	18.9	1.9	54%	35	8	77%
crc	98.7	2.5	84%	124	14	89%
fac	10.8	1.2	51%	19	4	79%
fibcall	5.1	1.6	41%	12	5	58%
fir	47.2	1.9	58%	79	9	89%
insertsort	7.8	2.1	31%	19	10	47%
loop3	3.7	1.5	39%	6	4	33%
matmult	27.3	5.6	56%	40	23	42%
minmax	1.8	1.1	9%	11	9	18%
ns	12.1	2.1	34%	30	13	57%
qsort-exam	101.7	1.9	78%	128	15	88%
qurt	97.2	1.4	75%	128	14	89%
select	94.6	1.8	72%	128	15	88%
sqrt	72.7	1.2	59%	128	14	89%

Table 4: Results (Cache-size: 1kB, Line-size: 8 Byte)

	average			upper bound		
	#UCB	#DC-UCB	diff	#UCB	#DC-UCB	diff
bs	7.6	1.2	55%	13	3	77%
bsort100	11.3	1.6	61%	20	5	75%
crc	58.4	1.9	83%	68	9	87%
fac	6.5	1.0	54%	11	2	82%
fibcall	3.9	1.3	48%	8	3	62%
fir	24.9	1.4	57%	41	5	88%
insertsort	4.3	1.6	29%	10	6	40%
loop3	2.2	1.1	32%	4	3	25%
matmult	15.4	3.1	61%	22	12	45%
minmax	1.3	0.9	8%	7	5	29%
ns	7.5	2.1	36%	17	7	59%
qsort-exam	65.2	1.5	79%	82	8	90%
qurt	171.5	1.1	75%	227	7	97%
select	52.6	1.3	75%	71	8	89%
sqrt	106.2	1.0	63%	190	7	96%

Table 5: Results (Cache-size: 4kB, Line-size: 16 Byte)

5.1 Discussion

The results show that our analysis (DC-UCBs) strongly outperforms the former approach (UCBs). Depending on the structure and the size of the task, the improvement of the maximal number of UCBs ranges from at least 18% (*minmax*) up to 97% (*qurt*, cache-size = 4KB, 8KB). In case the cache is small compared to the task-size (see Table 3), the number of UCBs is often bounded by the number of cache-sets (Cache-size 1kB, *qurt,select, sqrt*). So, the difference to the number of definitely-cached UCBs is smaller and improvements are less obvious.

Task *minmax* contains no loop and the useful cache blocks occur in a function invoked twice. Therefore, the number of (DC-)UCBs in both analyses is rather small and

[4]http://www.arm.com/products/CPUs/families/ARM7Family.html

[5]http://www.mrtc.mdh.se/projects/wcet/benchmarks.html

Task	bs	bsort100	crc	fac	fibcall	fir	insertsort	loop3	matmul	minmax	ns	qsort-exam	qurt	select	sqrt
#Instructions	69	123	288	48	47	209	81	1633	200	138	127	340	967	302	953
ratio 1KB	0.27	0.48	1.12	0.19	0.18	0.81	0.31	6.38	0.78	0.54	0.50	1.33	3.78	1.18	3.73
ratio 4KB	0.07	0.12	0.28	0.05	0.05	0.2	0.08	1.59	0.2	0.13	0.12	0.33	0.94	0.29	0.93
ratio 8KB	0.03	0.06	0.14	0.02	0.02	0.10	0.04	0.80	0.10	0.07	0.06	0.17	0.47	0.15	0.47

Table 3: Number of instructions and ratios of task-sizes to cache-sizes

	average			upper bound		
	#UCB	#DC-UCB	diff	#UCB	#DC-UCB	diff
bs	13.6	1.4	52%	24	5	79%
bsort100	18.9	1.9	54%	35	8	77%
crc	115.0	2.5	84%	134	14	90%
fac	10.8	1.2	51%	19	4	79%
fibcall	5.1	1.6	41%	12	5	58%
fir	47.8	1.9	58%	79	9	89%
insertsort	7.8	2.1	31%	19	10	47%
loop3	3.7	1.5	39%	6	4	33%
matmult	27.6	5.6	56%	40	23	42%
minmax	1.8	1.1	9%	11	9	18%
ns	12.9	2.4	35%	31	13	58%
qsort-exam	127.1	1.9	78%	160	15	91%
qurt	340.8	1.4	76%	449	14	97%
select	102.0	1.8	73%	138	15	89%
sqrt	204.1	1.2	60%	361	14	96%

Table 6: Results (Cache-size: 8kB, Line-size: 8 Byte)

both approaches differ only slightly. Best improvements are observed for programs that contain loops, recursive structures, or repeated invocation of routines. In these cases, the cache can work quite effectively which means that several cache blocks will be reused and memory accesses result in cache-hits. Lee over-approximates – in his notion of UCBs – the sets of these memory accesses. However, since the must-cache can only classify a subset of them as cache-hits, only a strongly reduced set is considered as definitely-cached UCBs. Of course, the improvement could even be more impressive, if we would have used a worse must-cache analysis. But due to virtual loop unrolling and virtual inlining (as mentioned before), we used a very precise must-cache analysis to derive realistic results.

A cache block typically contains more than one instruction. So, even for straight-line code sequences without loops, the sets of UCBs and DC-UCBs are not completely empty. Since nearly all programs contain such fragments with instructions executed at most once, the average number of (DC-)UCBs per instruction is reduced and so, also the average improvement per instruction (column 4) is always lower than the improvement on the maximal number. Nevertheless, values up to 80% are still possible. This indicates a large refinement of the CRPD also in case preemption is referred to a fixed set of preemption points.

To sum up, the difference between both notions can be explained due to difference between the sets of possibly-cached and definitely-cached memory blocks. To derive a general safe upper bound on the cache-related preemption delay the set of possibly-cached memory blocks must be considered. However, since the CRPD is always used in combination with the bound on the worst-case execution time, it is sufficient to consider the set of definitely-cached memory blocks. So,the DC-UCB analysis only accounts for cache-misses that are not taken into account by the timing analysis.

6 Conclusions

Prior useful cache block analyses use an over-approximation of the cache content to derive a safe over-approximation of the sets of UCBs. In combination with over-approximation of the timing analysis, several cache reloads may be counted twice.

In this paper, we have proposed an improved definition of useful cache block (UCB) namely the notion of definitely-cached UCB (DC-UCB). Instead of using an over-approximation, our approach uses an under-approximation of the cache content. Our definition requires a useful cache block to be definitely cached. In addition, we have shown the soundness of this notion when combined with the worst-case execution time bound. By using the same cache-analysis to derive the set of UCBs and the WCET bound, no cache-miss is accounted for in both analyses – but all cache-misses that might occur during preemptive execution are still taken into account. Furthermore, due to the tight coupling of DC-UCB and timing analysis the same DC-UCB analysis can be applied to set-associative caches and easily extended to an analysis for data caches even in case of array accesses.

The evaluation of our method, i.e., the comparison to the former approach, shows a strong improvement in the precision of the computed bound on the cache-related preemption delay. For all but two test-cases, the bound is reduced by at least 40% but also improvements up to more than 90% can be often observed.

As future work, we plan to investigate the improvement of our definition to the combined analysis of the preempting task and preempted task and to the worst-case response time analysis. A second point is the computation of lower bounds on the context switch costs at specific points. Such information may be useful for schedulability analyses.

Acknowledgement

We thank for Gernot Gebhard for his close collaboration during the implementation of the first prototype of the analysis. Furthermore, we thank Professor Reinhard Wilhelm for his comments and support writing this paper.

References

[1] J. V. Busquets-Mataix, J. J. Serrano, R. Ors, P. Gil, and A. Wellings. Adding instruction cache effect to schedulability analysis of preemptive real-time systems. In *Proceedings of the 2nd IEEE Real-Time Technology and Applications Symposium (RTAS'96)*, page 204, Washington, DC, USA, 1996. IEEE Computer Society.

[2] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252. ACM Press, New York, NY, 1977.

[3] C. Ferdinand, F. Martin, R. Wilhelm, and M. Alt. Cache behavior prediction by abstract interpretation. *Sci. Comput. Program.*, 35(2-3):163–189, 1999.

[4] J. Gustafsson. WCET challenge 2006. Technical report, Mälardalen University, January 2007.

[5] L. Ju, S. Chakraborty, and A. Roychoudhury. Accounting for cache-related preemption delay in dynamic priority schedulability analysis. In *Proceedings of the conference on Design, automation and test in Europe (DATE'07)*, pages 1623–1628, San Jose, CA, USA, 2007. EDA Consortium.

[6] S. kwan Kim, S. L. Min, and R. Ha. Efficient worst case timing analysis of data caching. In *IEEE Real-Time Technology and Applications Symposium (RTAS'96)*, pages 230–240. IEEE, 1996.

[7] C.-G. Lee, J. Hahn, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. S. Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. In *Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS'96)*, page 264, Washington, DC, USA, 1996. IEEE Computer Society.

[8] C.-G. Lee, J. Hahn, Y.-M. Seo, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. S. Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Trans. Comput.*, 47(6):700–713, 1998.

[9] S.-S. Lim, Y. H. Bae, G. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, S.-M. Moon, and C. S. Kim. An accurate worst case timing analysis for risc processors. *IEEE Trans. Softw. Eng.*, 21(7):593–604, 1995.

[10] T. Lundqvist and P. Stenström. Timing anomalies in dynamically scheduled microprocessors. In *Proceedings of the 20th IEEE Real-Time Systems Symposium (RTSS'99)*, page 12, Washington, DC, USA, 1999. IEEE Computer Society.

[11] F. Mueller. Compiler support for software-based cache partitioning. *SIGPLAN Not.*, 30(11):125–133, 1995.

[12] F. Mueller. Timing analysis for instruction caches. *Real-Time Systems*, 18:209–239, 2000.

[13] H. S. Negi, T. Mitra, and A. Roychoudhury. Accurate estimation of cache-related preemption delay. In *Proceedings of the 1st ACM international conference on Hardware/software codesign and system synthesis (CODES+ISSS'03)*, pages 201–206, New York, NY, USA, 2003. ACM.

[14] H. Ramaprasad and F. Mueller. Bounding preemption delay within data cache reference patterns for real-time tasks. In *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06)*, pages 71–80, Washington DC, 2006. IEEE Computer Society.

[15] J. Schneider. Cache and pipeline sensitive fixed priority scheduling for preemptive real-time systems. In *In Proceedings of the 21st IEEE Real-Time Systems Symposium (RTSS'2000)*, pages 195–204, 2000.

[16] J. Staschulat and R. Ernst. Multiple process execution in cache related preemption delay analysis. In *Proceedings of the 4th ACM international conference on Embedded software (EMSOFT'04)*, pages 278–286, New York, NY, USA, 2004. ACM.

[17] J. Staschulat and R. Ernst. Scalable precision cache analysis for preemptive scheduling. In *Proceedings of the 2005 ACM conference on Languages, compilers, and tools for embedded systems (LCTES'05)*, pages 157–165, New York, NY, USA, 2005. ACM.

[18] J. Staschulat and R. Ernst. Scalable precision cache analysis for real-time software. *Trans. on Embedded Computing Sys.*, 6(4):25, 2007.

[19] Y. Tan and V. Mooney. Integrated intra- and inter-task cache analysis for preemptive multi-tasking real-time systems. In *Proceedings of the 8th International Workshop SCOPES 2004, in: Lecture Notes on Computer Science, LNCS3199*, pages 182–199. Press, 2004.

[20] Y. Tan and V. Mooney. Timing analysis for preemptive multi-tasking real-time systems with caches. *Trans. on Embedded Computing Sys.*, 6(1):7, 2007.

[21] H. Theiling. Ilp-based interprocedural path analysis. In *In Proceedings of EMSOFT 2002, Second Workshop on Embedded Software*, 2002.

[22] H. Tomiyama and N. D. Dutt. Program path analysis to bound cache-related preemption delay in preemptive real-time systems. In *Proceedings of the 8th ACM international workshop on Hardware/software codesign (CODES'00)*, pages 67–71, New York, NY, USA, 2000. ACM.

[23] R. T. White, C. A. Healy, D. B. Whalley, F. Mueller, and M. G. Harmon. Timing analysis for data caches and set-associative caches. In *Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium (RTAS'97)*, page 192, Washington, DC, USA, 1997. IEEE Computer Society.

[24] A. Wolfe. Software-based cache partitioning for real-time applications. *J. Comput. Softw. Eng.*, 2(3):315–327, 1994.